Enseigner «Frama-C pour la cybersécurité» : retour d'expérience

Julien Signoles
Université Paris-Saclay, CEA, List
Palaiseau, France
julien.signoles@cea.fr

***Résumé*—Cet article présente un retour d'expérience lié à un enseignement de Frama-C pour la cybersécurité dispensé dans plusieurs formations d'ingénieurs et universitaires de niveau master. Il est constitué d'un cours et d'un TP de trois heures chacun. Frama-C est une plateforme *open source* fournissant des analyseurs de code C. Le cours est articulé autour de ses trois techniques et analyseurs principaux, à savoir la vérification déductive avec Wp, l'interprétation abstraite avec Eva, et la vérification d'annotations à l'exécution avec E-ACSL. Le but du cours est de faire découvrir ces techniques formelles aux étudiants en insistant sur leurs aspects pratiques en cybersécurité.**

I. Introduction

Frama-C est une plateforme *open source* d'analyse de code C [2], [24] développée depuis 2005 et aujourd'hui bien établie. Elle permet d'être aussi bien utilisée par le monde académique que déployée pour des usages industriels. En particulier, Frama-C a été initialement utilisée pour vérifier des propriétés de sûreté pour des applications embarquées critiques, par exemple pour l'avionique [7], le nucléaire [31], ou le spatial [9], et est aujourd'hui aussi utilisée pour vérifier des propriétés de sécurité. Par exemple, Thales a récemment certifié aux niveaux EAL6-EAL7 des Critères Communs une machine virtuelle JavaCard à l'aide d'une méthodologie reposant sur Frama-C et reconnue par l'ANSSI [14]–[16].

Depuis de nombreuses années, Frama-C est utilisée pour l'enseignement [1]. Dans ce cadre, elle est en général utilisée lors de séances de Travaux Pratiques (TP), au sein d'enseignements liés aux méthodes formelles, comme la vérification déductive [20], l'interprétation abstraite [12] ou la vérification d'annotations à l'exécution [21]. Quelques usages dans des formations en cybersécurité sont aussi avérés, par exemple au master cybersécurité de l'Université Grenoble Alpes [2]. Malgré tout, les retours d'expérience demeurent rares [13], [17], [19], [35]. En particulier, aucun d'eux n'aborde la vérification d'annotations à l'exécution et ne met l'accent sur les usages pour la cybersécurité. Quelques publications [5], [23], [25], [26] présentent également des tutoriaux. La deuxième est limitée à la vérification déductive, la troisième à la vérification d'annotations à l'exécution, tandis que la première combine les deux. La quatrième aborde la génération de tests, qui n'est pas abordée dans l'enseignement dont il est question ici et est donc complémentaire. En résumé, aucun retour d'expérience n'a encore été fait concernant l'usage de Frama-C dans des modules d'enseignement en cybersécurité et incluant les trois principales techniques de vérification incluses dans la plateforme, à savoir la vérification déductive, l'interprétation abstraite et la vérification d'annotations à l'exécution.

Cet article vise à palier ce manque en restituant l'expérience acquise en enseignant une introduction à «Frama-C pour la cybersécurité», qui prend la forme d'un cours et d'un TP, de trois heures chacun dans leurs versions nominales et d'une heure et demie dans leurs versions courtes. Elle a été créée à la demande de Telecom SudParis fin 2022. En 2025, elle est dispensée dans trois écoles d'ingénieurs (Telecom SudParis, CentraleSupelec Rennes et École des Mines Paris) et deux masters de l'Université Bretagne Sud (masters «*Cyberus Erasmus mundus joint master in cybersecurity*» et «Cyber-Sécurité des Systèmes Embarqués»), contribuant ainsi à former aujourd'hui une petite centaine d'étudiants par an. Elle est intégrée dans des modules d'enseignement de filières cybersécurité ou systèmes embarqués.

Le plan de cet article est le suivant. D'abord, la section II introduit brièvement la plateforme Frama-C. Ensuite, les sections III et IV présentent respectivement le cours et son TP. Enfin, la section V dresse un bilan de cette expérience et quelques pistes d'amélioration.

II. Frama-C, en bref

Frama-C est une plateforme *open source* d'analyse de code C. Une des forces de Frama-C, mais qui contribue également à la rendre difficile à prendre en main, est qu'il ne s'agit pas d'un unique outil d'analyse de code source, mais plutôt d'une large collection d'analyseurs de code fournis comme des greffons tous connectés au noyau de la plateforme [28] et fournissant chacun des fonctionalités différentes. Un certain nombre d'éléments de la plateforme sont ainsi partagés entre tous les analyseurs, grâce au noyau. En particulier, les interactions avec l'utilisateur se font toujours d'une manière similaire, que ce soit *via* l'interface graphique ou en ligne de commandes dans un terminal. Un autre point essentiel est que le code C analysé peut être étendu par des annotations formelles écrites dans le langage ACSL [6],

Ce travail a bénéficié d'une aide de l'état gérée par l'Agence Nationale de la Recherche au titre de France 2030 portant la référence ANR-23-CMAS-0014. Les opinions exprimées dans ce document ne reflètent pas nécessairement l'avis du gouvernement français.

1. Quelques liens vers des cours existants sont mentionnés sur la très incomplète page https://www.frama-c.com/html/teaching.html.

2. https://im2ag-moodle.univ-grenoble-alpes.fr/pluginfile.php/69164/course/section/10508/slides_framac.pdf.

langua franca de la plateforme. Enfin, la plateforme permet la collaboration entre ses analyseurs [11], ce qui renforce ses capacités d'analyse pour des cas d'usage avancés [22].

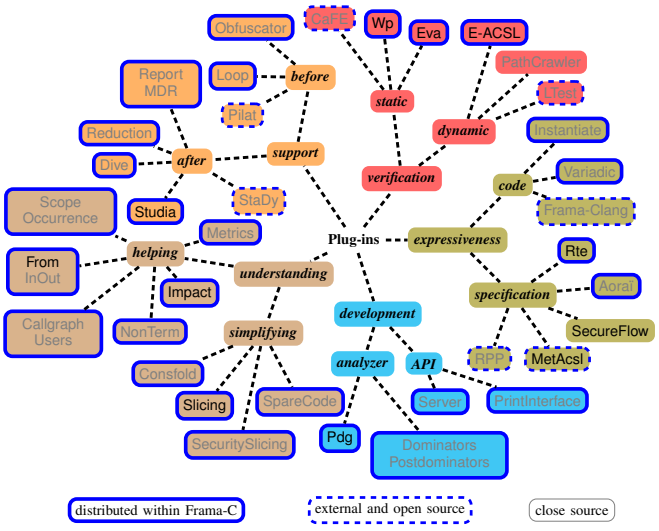

FIGURE 1. Vue des analyseurs de Frama-C issue d'un des transparents du cours : ceux grisés ne sont pas évoqués en cours.

La figure II, issue d'un des transparents du cours, présente un large éventail des greffons Frama-C existants. La plupart sont intégrés à la distribution standard de Frama-C, alors que certains autres sont distribués à l'extérieur et d'autres encore sont *close source*. La figure distingue cinq catégories de greffons. Les vérificateurs sont dédiés à vérifier effectivement des propriétés de programmes. Ils incluent en particulier les trois principaux greffons de la plateforme, à savoir Wp [4] reposant sur la vérification déductive de programmes, Eva [8] reposant sur l'interprétation abstraite et E-ACSL [3] reposant sur la vérification d'annotations à l'exécution. Certains greffons permettent d'améliorer l'expressivité des programmes analysés ou des propriétés vérifiées. Par exemple Frama-Clang supporte (un sous-ensemble de) C++, et Aorai permet d'exprimer des propriétés temporelles. Quelques greffons fournissent des services aux autres greffons *via* leur API, par exemple un graphe de dépendance de programmes (PDG pour *Program Dependence Graph* [30]). Plusieurs analyseurs permettent à l'utilisateur de mieux comprendre le code analysé, soit en simplifiant ce dernier comme la propagation de constantes avec Constfold, soit en calculant des informations spécifiques comme les entrées et les sorties des fonctions avec Inout. Enfin, quelques greffons facilitent l'utilisation d'autres analyseurs de la plateforme, soit en amont par exemple en obfuscant un code sensible avec Obfuscator ou en aval par exemple en proposant des sorties récapitulant les résultats des analyses effectuée dans des formats divers avec Report.

III. CONTENU DU COURS

Le cours [3] est d'une durée de trois heures. Il existe également une version courte d'une heure et demie [4]. Ce format n'est pas adapté à un apprentissage en profondeur de Frama-C, ni même d'un analyseur particulier. Par conséquent, le parti pris est de présenter aux étudiants les usages possibles de ces techniques, tout en mettant le doigt sur le fait qu'elles peuvent être complexes à maîtriser. En revanche, son but n'est pas de former de futurs experts des différentes techniques formelles. En particulier, contrairement à un module d'enseignement standard consacrant plus de vingt heures à une techniques formelle spécifique, comme la vérification déductive ou l'interprétation abstraite, existant dans des formations spécialisées (par exemple, le master MPRI [5]), cet enseignement ne présente pas les théories sous-jacentes. Le seul pré-requis nécessaire est une connaissance minimale du langage C, qui soit suffisante pour comprendre des petits programmes avec pointeurs et allocations dynamiques (en TP), sans avoir besoin de coder.

Le cours est divisé en six parties. La première partie est un préambule mettant le cours en contexte en justifiant le besoin d'avoir recours à des solutions outillées reposants sur des méthodes formelles pour améliorer la confiance envers des logiciels. Elle introduit aussi informellement des qualificatifs fondamentaux pour une analyse de code, à savoir ses caractères *statique*, *automatique* (et rapide), *exacte* et *correcte*, tout en explicitant le fait que définir une analyse avec l'ensemble ces caractéristiques n'est pas possible en vertu du théorème de Rice [33]. Cela permet d'introduire rapidement différentes techniques formelles qui relâchent nécessairement au moins une des caractéristiques ci-dessus : la vérification d'annotations à l'exécution [21], la vérification de modèle [18], [32], l'interprétation abstraite [12], et la vérification déductive [20].

Après ce préambule, le plan du cours est annoncé avec une partie dédiée à une courte présentation de Frama-C dans son ensemble, trois parties dédiées chacunes aux trois principales techniques de vérification de la plateforme, à savoir vérification déductive avec Wp, interprétation abstraite avec Eva et vérification d'annotations à l'exécution avec E-ACSL, dans cet ordre, et une dernière partie dédiée à des usages plus avancés de Frama-C pour vérifier des propriétés de sécurité spécifiques. On peut constater qu'aucune partie n'est dédiée à la vérification de modèle : ceci vient du fait que cette technique est très peu développée aujourd'hui au sein de Frama-C. L'ordre de présentation des trois principaux analyseurs n'est pas aléatoire : les techniques formelles sont présentées de la plus complexe à la plus simple d'utilisation. Cela permet de passer plus de temps sur la première à un moment où étudiants et enseignant sont encore frais et d'aller plus rapidement sur les deux autres. Notamment, il est possible d'aller très vite sur la partie E-ACSL si jamais le temps manque. Les trois premières parties (contexte, Frama-C et Wp) sont abordées dans la première moitié du cours, et les trois dernières (Eva, E-ACSL et usages avancés) dans la seconde.

La seconde partie introduit Frama-C dans son ensemble, en insistant sur le choix fait par la plateforme de ne pas proposer une seule technique formelle, mais de miser plutôt sur leurs variétés et leurs possibles combinaisons. Cette partie présente

3. https://julien-signoles.fr/teaching/slides-frama-c-cyber.pdf
4. https://julien-signoles.fr/teaching/slides-frama-c-cyber-short.pdf
5. https://wikimpri.dptinfo.ens-cachan.fr/doku.php

notamment la figure II, principalement pour préciser qu'aucun des greffons grisés n'est abordé en cours ou en TP : au-delà des trois greffons principaux, les autres greffons non grisés sont évoqués dans la dernière partie du cours présentant les usages avancés, sauf Studia qui est utilisé en TP.

Le plan des quatre parties «techniques» est délibérément toujours similaire : d'abord fournir une intuition de la technique sous-jacente, sans aborder la théorie correspondante, ensuite, au travers d'exemples, présenter les spécificités en termes d'usage en insistant sur son utilité principale et les principales difficultés pratiques de mises en œuvre, et enfin illustrer la technique à travers une démo et (au moins) un exemple d'usage industriel réel. Ce choix permet d'assurer une cohérence globale du cours, tout en rendant plus concrètes les techniques abordées. En outre, tous les exemples et toutes les démos sont réalisés en direct devant eux avec Frama-C, ce qui permet de prendre un peu d'avance sur des aspects qu'ils (re)verront en TP et, surtout, de rendre le cours plus vivant. Le retour des étudiants que j'ai pu obtenir semble confirmer que combiner des petits exemples et des démos en direct avec des cas d'usages industriels réels est très bien reçu de leur part. Un autre avantage est lié à la gestion du temps, puisqu'il est facile de rattraper un retard éventuel, en omettant la présenation en direct de tel ou tel démo ou exemple. Ces derniers sont en particulier intégrés, avec leur correction, aux transparents du cours, ce qui permet si nécessaire de présenter directement la solution sans avoir à le réaliser en direct. En outre, cela permet d'offrir un support de cours auto-contenu aux étudiants.

Le cours insiste également sur les différences pratiques entre les trois techniques, aussi bien concernant leurs avantages que leurs inconvénients. Ainsi, la vérification déductive permet de vérifier des propriétés complexes au prix d'une absence d'automatisation totale : l'utilisateur doit ajouter manuellement des annotations, tandis qu'une partie des preuves automatiques peuvent échouées. Ici, le fait que l'enseignant effectue lui-même et en direct plusieurs exemples de difficulté progressive permet de montrer la difficulté de la preuve de programmes aux étudiants, sans perdre de temps à les laisser chercher des solutions pour lesquelles ils ne sont pas bien armés pour les trouver rapidement. L'interprétation abstraite est présentée plus rapidement puisque les étudiants manipulent ensuite l'outil eux-même en TP : le cours montre que ce type d'outils est adapté pour trouver des comportements indéfinis pouvant correspondre à des vulnérabilités dans le code, tout en pouvant garantir leur absence lorsqu'aucune alarme est levée. Le cours insiste néanmoins sur le fait que, même si la technique est automatique, elle nécessite une expertise de l'utilisateur pour régler correctement ses paramètres afin d'obtenir un bon compromis entre précision, temps d'exécution et consommation mémoire. La vérification d'annotations à l'exécution est abordée plus rapidement, puisque cette technique est totalement automatique et ne nécessite pas d'expertise particulière de la part de l'utilisateur. Le cours insiste néanmoins sur le fait qu'un environnement d'exécution concret ou virtuel est nécessaire et que les surcoûts en temps d'exécution et en consommation mémoire peuvent être importants en pratique.

La dernière partie du cours présente plusieurs usages avancées utilisés en contexte cyber. Le premier concerne la combinaison d'au moins deux des trois vérificateurs principaux présentés auparavant, pour vérifier l'absence de comportements indéfinis notamment. Le second usage combine Eva ou E-ACSL avec le greffon SecureFlow [1] dédié à la vérification de non-interférence et pouvant aussi détecter certaines attaques par canaux temporels. Le dernier usage présenté combine Wp ou E-ACSL avec le greffon MetAcsl [34] pour vérifier des propriétés d'intégrité et de confidentialité, typiquement pour des systèmes d'exploitation. Dans chacun des cas, l'idée est principalement de présenter ce qu'il est possible de faire, sans donner trop de détails techniques sur le moyen d'y parvenir.

La version réduite du cours conserve le plan en six parties, en réduisant chacune. Les points essentiels demeurent présents mais sont moins répétés, ce qui peut nuire à leur bonne intégration par les étudiants.

IV. Travaux Pratiques

La séance de travaux pratiques (TP) [6] est, tout comme le cours, d'une durée de trois heures, tout en existant aussi en version réduite d'une heure et demie [7]. Son but principal est de voir les possibilités offertes par Frama-C, et notamment par le greffon Eva dédié à l'analyse de valeurs par interprétation abstraite. Il s'agit aussi de toucher du doigt le fait que ce type d'outils, certes puissants, nécessite d'être paramétré finement afin d'obtenir les résultats souhaités.

Le TP est organisé en deux parties, la première dédiée à Eva et la seconde dédiée à Wp. Néanmoins, seule la première est en réalité l'objet du TP, la seconde étant présente uniquement pour fournir quelques exercices supplémentaires pour les très rares étudiants suffisamment motivés pour faire d'eux-mêmes des exercices facultatifs à la maison. Chaque partie est constituée de deux exercices indépendants, le premier — très facile— aidant à prendre en main l'outil et le second constituant le problème réel. Chaque exercice est subdivisé en questions permettant aux étudiants d'avancer pas à pas. La version courte du TP présente une version simplifiée du second exercice sur Eva, et n'est pas détaillée ici. Tout comme pour le cours, son temps réduit laisse néanmoins moins de temps aux étudiants pour bien intégrer les notions abordées.

La première partie du TP est dédiée à Eva. La difficulté liée à la création d'un sujet de TP autour d'un outil d'interprétation abstraite comme Eva vient du fait que l'analyse est totalement automatique si le code C analysé est trop facile, ce qui rend l'exercice inintéressant, mais que l'analyse devient rapidement difficile à paramétrer précisément dès lors que le code C analysé est un tant soit peu réaliste. Trouver un code C à analyser qui soit à la fois faisable et compréhensible en quelques heures par des étudiants inexpérimentés, tout en étant intéressant n'est donc pas simple. Le parti pris ici est de guider les étudiants pas à pas en leur faisant notamment faire des observations sur les résultats émis par l'analyseur. En pratique,

6. https://julien-signoles.fr/teaching/tp-cyber/tp.md
7. https://julien-signoles.fr/teaching/tp-cyber/short/tp-short.md

il est nécessaire de les aider en répondant individuellement à leur question en séance mais, dans ces conditions, la majorité des étudiants parviennent bien à comprendre tout en réalisant la majeure partie du TP dans les trois heures [8]. Il est à noter que les réponses ne sont pas en ligne et que les LLM ne sont à ce jour d'aucune aide pour résoudre cette partie du TP.

Le premier exercice repose sur un petit programme issu de la suite de tests *Juliet* [9] du NIST visant à éprouver les outils d'analyse de code. L'exercice consiste à utiliser Eva pour trouver le comportement indéfini dans le code. À travers deux questions, il permet aux étudiants de prendre en main l'outil sur un code simple en observant les résultats de l'analyse non paramétrée puis d'améliorer leur précision en utilisant le paramètre principal de l'analyseur, à savoir l'option `-eva-slevel <n>` qui offre un moyen simple de partionnement de traces [29], une des principales techniques existantes pour améliorer la précision des analyseurs abstraits. Une fois un bon niveau de partionnement indiqué, l'analyseur trouve de manière certaine un comportement indéfini. Grâce à ce premier exercice, les étudiants touchent du doigt les imprécisions de ce type d'analyseur lorsqu'il est mal paramétré.

Le second exercice constitue le sujet principal du TP. Il s'agit d'analyser un module de chiffrement/déchiffrement d'un code de Bacon provenant du projet *Rosetta Code* [10], dans lequel quelques erreurs ont été ajoutées. Le but de l'exercice est d'utiliser Eva pour trouver les erreurs, de les corriger en ajoutant ou modifiant une ligne de code pour chacune, puis de montrer qu'il n'y a aucun comportement indéfini dans le programme pour un large ensemble d'entrées. Il est subdivisé en cinq questions. La première consiste à lancer l'analyse sans paramétrage particulier et de se concentrer sur une alarme spécifique parmi celles levées. Les étudiants doivent alors observer les valeurs abstraites importantes calculées par l'analyseur, utiliser le greffon Studia pour naviguer dans le code et trouver l'origine de l'alarme [27], comprendre cette dernière et corriger le code en conséquence pour supprimer l'alarme. La seconde question suit le même modèle, même si les étudiants observent le code *via* un autre mécanisme fourni par Eva et doivent cette fois améliorer la précision de l'analyse pour enlever l'alarme considérée, qui est fausse. Cette question est aussi l'occasion de leur faire observer le calcul de point-fixe effectué par Eva lorsqu'elle analyse une boucle. La troisième question demande aux étudiants d'enlever toutes les autres alarmes en améliorant la précision de l'analyse et en corrigeant le code quand nécessaire : elle permet aux étudiants de répéter par eux-même la méthodologie sous-jacente aux deux premières questions, tout en découvrant un paramètre supplémentaire de l'analyse (`-eva-auto-loop-unroll <n>`, qui améliore la précision en déroulant les boucles n fois). La quatrième question est d'une nature différente, puisqu'elle demande aux étudiants de généraliser le contexte d'entrée de l'analyse en utilisant une primitive fournie par Eva pour cela : alors que les trois premières questions analysaient en fait le programme pour un message fixé, cette question permet d'étendre le contexte d'entrée à une large classe de messages. La cinquième question est similaire et étend encore davantage la classe de messages considérée. Elle permet en outre de découvrir un paramètre supplémentaire de l'analyse (`-ilevel <n>`, qui fixe le nombre n d'éléments jusqu'auquel les ensembles sont représentées par des ensembles discrets de n valeurs et non approximés par des intervalles comprenant tous les nombres entre les bornes inférieure et supérieure de l'ensemble). Partir d'une entrée précise et généraliser ainsi peu à peu le contexte d'entrée est une méthodologie classique d'utilisation d'Eva.

Les deux exercices proposés dans la seconde partie, optionnelle, sur Wp sont très classiques : le premier consiste à prouver un programme codant une multiplication $x \times y$ comme x additions du nombre y à partir de 0, tandis que le second vise à vérifier l'implémentation d'une recherche dichotomique dans un tableau trié. L'absence de comportement indéfini n'est prouvé que dans le second exercice : la condition garantissant l'absence de débordement arithmétique dans le premier exercice est compliquée à inférer, sans être particulièrement intéressante. Je ne détaille pas plus avant cette seconde partie, classique et très rarement réalisée par les étudiants dans ce TP.

V. Bilan et perspectives

Le bilan de cet enseignement à Frama-C pour la cybersécurité, introductif et résolument orienté vers les aspects pratiques des méthodes formelles, est très positif dans sa version standard de deux fois trois heures. Les étudiants semblent notamment particulièrement appréciés les exemples concrets du cours et les applications industrielles présentées. Leurs questions sont souvent pertinentes et permettent d'aborder des points ne figurant pas sur les transparents, notamment concernant la sémantique du C ou les applications dans l'industrie. Un effet positif est de rappeler (voire d'apprendre) aux étudiants certains éléments de programmation sécurisée en C, comme la notion de comportement indéfini, l'arithmétique entière bornée, les tableaux dynamiques et l'arithmétique de pointeurs, le fait que les fonctions d'allocation dynamiques comme `malloc` peuvent retourner la valeur `NULL` et les mécanismes de gestion d'erreurs.

Les améliorations possibles portent surtout sur le TP, dont l'énoncé pourrait être encore plus détaillé pour pouvoir être réalisé davantage en autonomie. En outre, il doit être adapté à Ivette, la nouvelle interface graphique de Frama-C [10].

Remerciements

Je souhaiterais remercier Virgile Prevosto, à l'origine de l'idée originale du TP, et Nikolai Kosmatov, auteur de certains exemples du cours. Par ailleurs, cet article n'aurait pas vu le jour si des responsables de formation ne m'avait pas donné l'opportunité de dispenser cet enseignement dans leurs établissements, notamment Olivier Levillain à Telecom SudParis, Jean-François Lalande et Pierre Wilke à CentraleSupelec Rennes, Salah Sadou et Guy Gogniat à l'Université Bretagne Sud et Olivier Hermant à l'École des Mines Paris.

8. Lorsque le TP est évalué, un délai d'une semaine est laissé pour le finir.
9. https://samate.nist.gov/SARD/test-suites/112
10. https://rosettacode.org/wiki/Bacon_cipher

RÉFÉRENCES

[1] Gergö Barany and Julien Signoles. Hybrid Information Flow Analysis for Real-World C Code. In *Tests and Proofs (TAP)*, 2017. https://https://doi.org/10.1007/978-3-319-61467-0_2.

[2] Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolaï Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles, and Nicky Williams. The Dogged Pursuit of Bug-Free C Programs : The Frama-C Software Analysis Platform. *Communications of the ACM*, 2021. https://doi.org/10.1145/3470569.

[3] Thibaut Benjamin and Julien Signoles. *Runtime Annotation Checking with Frama-C : The E-ACSL Plug-in*. In [24], July 2024. https://doi.org/10.1007/978-3-031-55608-1_5.

[4] Allan Blanchard, François Bobot, Patrick Baudin, and Loïc Correnson. *Formally Verifying that a Program Does What It Should : The Wp Plug-in*. In [24], July 2024. https://doi.org/10.1007/978-3-031-55608-1_4.

[5] Allan Blanchard, Nikolai Kosmatov, and Frédéric Loulergue. A Lesson on Verification of IoT Software with Frama-C. In *International Conference on High Performance Computing & Simulation (HPCS)*, 2018. https://doi.org/10.1109/HPCS.2018.00018.

[6] Allan Blanchard, Claude Marché, and Virgile Prevosto. *Formally Expressing what a Program Should Do : The ACSL Language*. In [24], July 2024. https://doi.org/10.1007/978-3-031-55608-1_1.

[7] Abderrahmane Brahmi, Marie-Jo Carolus, David Delmas, Mohamed Habib Essoussi, Pascal Lacabanne, Victoria Moya Lamiel, Famantanantsoa Randimbivololona, and Jean Souyris. Industrial use of a safe and efficient formal method based software engineering process in avionics. In *European Conference on Embedded Real Time Software and Systems (ERTS)*, 2020. https://www.erts2020.org/uploads/presentation-pdf/th1-b3-souyris.pdf.

[8] David Bühler, André Maroneze, and Valentin Perrelle. *Abstract Interpretation with the Eva Plug-in*. In [24], July 2024. https://doi.org/10.1007/978-3-031-55608-1_3.

[9] Rovedy Aparecida Busquim e Silva, Nanci Naomi Arai, Luciana Akemi Burgareli, Jose Maria Parente de Oliveira, and Jorge Sousa Pinto. *Exploring Frama-C Resources by Verifying Space Software*. In [24], July 2024. https://doi.org/10.1007/978-3-031-55608-1_14.

[10] Loïc Correnson. Ivette : A Modern GUI for Frama-C. In *Workshop on Formal Integrated Development Environment (F-IDE)*, 2023. https://doi.org/10.1007/978-3-031-26236-4_10.

[11] Loïc Correnson and Julien Signoles. Combining Analyses for C Program Verification. In *Formal Methods for Industrial Case Studies (FMICS)*, 2012. https://doi.org/10.1007/978-3-642-32469-7_8.

[12] Patrick Cousot. *Principles of Abstract Interpretation*. MIT Press, 2022.

[13] Léo Creuse, Claire Dross, Christophe Garion, Jérôme Hugues, and Joffrey Huguet. Teaching Deductive Verification Through Frama-C and SPARK for Non Computer Scientists. In *Formal Methods Teaching Workshop (FMTea)*, 2019. https://doi.org/10.1007/978-3-030-32441-4_2.

[14] Adel Djoudi, Martin Hána, and Nikolaï Kosmatov. Formal Verification of a JavaCard Virtual Machine with Frama-C. In *International Conference on Formal Methods (FM)*, 2021. https://link.springer.com/chapter/10.1007/978-3-030-90870-6_23.

[15] Adel Djoudi, Martin Hána, and Nikolaï Kosmatov. *Proof of Security Properties : Application to JavaCard Virtual Machine*. In [24], July 2024. https://doi.org/10.1007/978-3-031-55608-1_16.

[16] Adel Djoudi, Martin Hána, Nikolai Kosmatov, Milan Kříženecký, Franck Ohayon, Patricia Mouy, Arnaud Fontaine, and David Feliot. A Bottom-Up Formal Verification Approach for Common Criteria Certification : Application to JavaCard virtual machine. In *European Congress on Embedded Real-Time Systems (ERTS)*, 2022. https://hal.science/hal-03695829.

[17] Catherine Dubois, Virgile Prevosto, and Guillaume Burel. Teaching Formal Methods to Future Engineers. In *Formal Methods Teaching Workshop (FMTea)*, 2019. https://doi.org/10.1007/978-3-030-32441-4_5.

[18] E. Allen Emerson and Edmund M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. *Automata, Languages and Programming*, 1980. https://doi.org/10.1007/3-540-10003-2_69.

[19] Matthias Güdemann. Online Teaching of Verification of C Programs in Applied Computer Science. In *Formal Methods Teaching Workshop (FMTea)*, 2021. https://doi.org/10.1007/978-3-030-91550-6_2.

[20] Reiner Hähnle and Marieke Huisman. *Deductive Software Verification : From Pen-and-Paper Proofs to Industrial Tools*. 2019.

[21] Marieke Huisman and Anton Wijs. Runtime annotation checking. In *Concise Guide to Software Verification*, Texts in Computer Science. Springer, August 2023. https://doi.org/10.1007/978-3-031-30167-4.

[22] Nikolai Kosmatov, Artjom Plaunov, Subash Shankar, and Julien Signoles. *Combining Analyses within Frama-C*. In [24], July 2024. https://doi.org/10.1007/978-3-031-55608-1_9.

[23] Nikolaï Kosmatov, Virgile Prevosto, and Julien Signoles. A Lesson on Proof of Programs with Frama-C. Invited Tutorial Paper. In *International Conference on Tests and Proofs (TAP)*. https://doi.org/10.1007/978-3-642-3.

[24] Nikolaï Kosmatov, Virgile Prevosto, and Julien Signoles. *Guide to Software Verification with Frama-C : Core Components, Usages and Applications*. Springer Cham, July 2024. https://link.springer.com/book/10.1007/978-3-031-55608-1.

[25] Nikolaï Kosmatov and Julien Signoles. A lesson on runtime assertion checking with Frama-C. In *International Conference on Runtime Verification (RV)*, 2013. https://doi.org/10.1007/978-3-642-40787-1_29.

[26] Nikolai Kosmatov, Nicky Williams, Bernard Botella, Muriel Roger, and Omar Chebaro. A lesson on structural testing with pathcrawler-online.com. In *International Conference on Tests and Proofs (TAP)*, 2012. https://doi.org/10.1007/978-3-642-30473-6_15.

[27] André Maroneze and Valentin Perrelle. *Tools for Program Understanding*. In [24], July 2024. https://doi.org/10.1007/978-3-031-55608-1_8.

[28] André Maroneze, Virgile Prevosto, and Julien Signoles. *The Heart of Frama-C : The Frama-C Kernel*. In [24], July 2024. https://doi.org/10.1007/978-3-031-55608-1_2.

[29] Laurent Mauborgne and Xavier Rival. Trace Partitioning in Abstract Interpretation Based Static Analyzers. 2005. https://doi.org/10.1007/978-3-540-31987-0_2.

[30] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *Software Engineering Symposium on Practical Software Development Environments*, 1984. https://doi.org/10.1145/800020.808263.

[31] Alain Ourghanlian. Evaluation of static analysis tools used to assess software important to nuclear power plant safety. *Nuclear Engineering Technology*, 2015. https://doi.org/10.1016/j.net.2014.12.009.

[32] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming*, volume 137, 1982. https://https://doi.org/10.1007/3-540-11494-7_22.

[33] Henry G. Rice. Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical Society*, 1953. https://doi.org/10.2307/1990888.

[34] Virgile Robles, Nikolai Kosmatov, Virgile Prevosto, Louis Rilling, and Pascale Le Gall. Metacsl : Specification and verification of high-level properties. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2019. https://doi.org/10.1007/978-3-030-17462-0_22.

[35] Salwa Souaf and Frédéric Loulergue. Experience report : Teaching code analysis and verification using Frama-C. In *International Workshop on Applicable Formal Methods (appFM)*, 2021. https://inria.hal.science/hal-03338928.